[Modularity Convention](https://wiki.eisgroup.com/display/GRC/Modularity+Convention)

# Introduction

The goal of this document is to develop recommendations to EIS Conventions to enforce modularity in any EIS project.

# Definitions

## **Module**

A software module is a deployable, manageable, natively reusable, composable, stateless unit of software that provides a concise interface to consumers.

In EIS projects JAR file is the unit of modularity. So in Runtime Model "module" means a JAR file, in Development Model "module" means Maven module with <packaging>jar</packaging>.

## **API**

Public part of a module. Contains artifacts that module's consumers can import and use. API should be contained in packages that have **api** parent package.

## **IMPL**

Private part of a module. Contains implementation of the module's API. It cannot be used by any consumer directly. IMPL should be contained in packages that have **impl** parent package.

## **Dependencies between modules**

We say that module A depends on module B only if module A uses any artifact from module B. In Java it means module A imports at least one class from module B.

# Conventions

## **API and IMPL should be explicitly defined**

Public and private parts of every module should be explicitly defined by module and package structures. Current version of Java doesn't support explicit definition of public/private parts of a module, so we need to use following two types of modules structures and package naming:

### Monolith module

Monolith module contains both API and IMPL parts. It has the following structure:

Graphical user interface, text, application

Description automatically generated

where 'api' package is an API part of the module and 'impl' package is a IMPL part of the module.

#### Pros

* Easy to use, because there is only one Maven project

#### Cons

* Hard to reuse, because even if you want to do total replacement of API implementation, default implementation will always be with you, with all its beans in Spring context and all external dependencies.
* We cannot forbid to use/extend classes from IMPL part - from JLS perspective it would be correct syntax.

#### Use Cases

* If you implement some feature and you really don't have clear vision how this feature will be separated to modules and how it will be customized, you can start from Monolith modules and then move to Multi-parted modules.
* Utility modules.

### Multi-parted module

Multi-parted module is in a fact two modules that implement [Separate Abstraction pattern](https://dzone.com/refcardz/patterns-modular-architecture): it places API and IMPL parts of a Monolith module to different modules:

A picture containing timeline

Description automatically generated

where:

* module-name - Maven project with POM packaging
* module-name-api - API part of the module. Be aware that despite of placing API in separate module we still should use **api** package here.
* module-name-impl - IMPL part of the module. Be aware that despite of placing IMPL in separate module we still should use **impl** package here.

#### Pros

* Easy to reuse.
* Easy to forbid use/extend classes from IMPL.

#### Cons

* Hard to use, because we have two modules instead of one and should manage this situation.

#### Use Cases

* Because most EIS projects are software that is meant to be customized by extending customization points, in [Use/Reuse paradox](https://dzone.com/articles/usereuse-paradox) we need to choose 'Reuse' part. So Multi-parted modules should be used as default design in most cases.

## **Dependencies resolving and directions**

Dependencies should be directed and resolved in the following way:

1. API modules depend on other API modules only. **Rule for automatic checking of \*.java files:** any file in **api** package can import only elements from other **api** packages.
2. IMPL modules depend on API modules only. **Rule for automatic checking of \*.java files:** any file in **impl** package can import elements from other **api** packages and from the same **impl** package.
3. Cyclic dependencies are forbidden.
4. We use IoC to inject an API implementation.

## **Modules should be levelized**

To provide clear separation of concerns and support predictable dependency management, we need to separate different layers functionality to different modules.

Let's consider the following example. We want to implement some feature (for example, new product in PAS). In this implementation we need to read/write data from/to storage, run business logic, provide access to the feature via services and add UI to work with the feature. So we have the following layers here:

Table

Description automatically generated

And the simple (and incorrect) solution here would be creating one module and try to separate layers by packages:

Chart

Description automatically generated

to have the following Layers-to-Modules dependencies:

Diagram, table

Description automatically generated

Correct solution here would be using several modules to create the following Layers-to-Modules dependencies:

Diagram

Description automatically generated